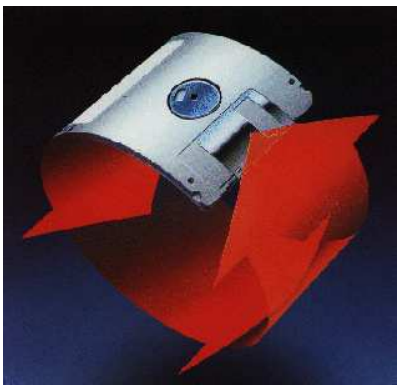
# SOUR Modules:
# Comparator & Modifier

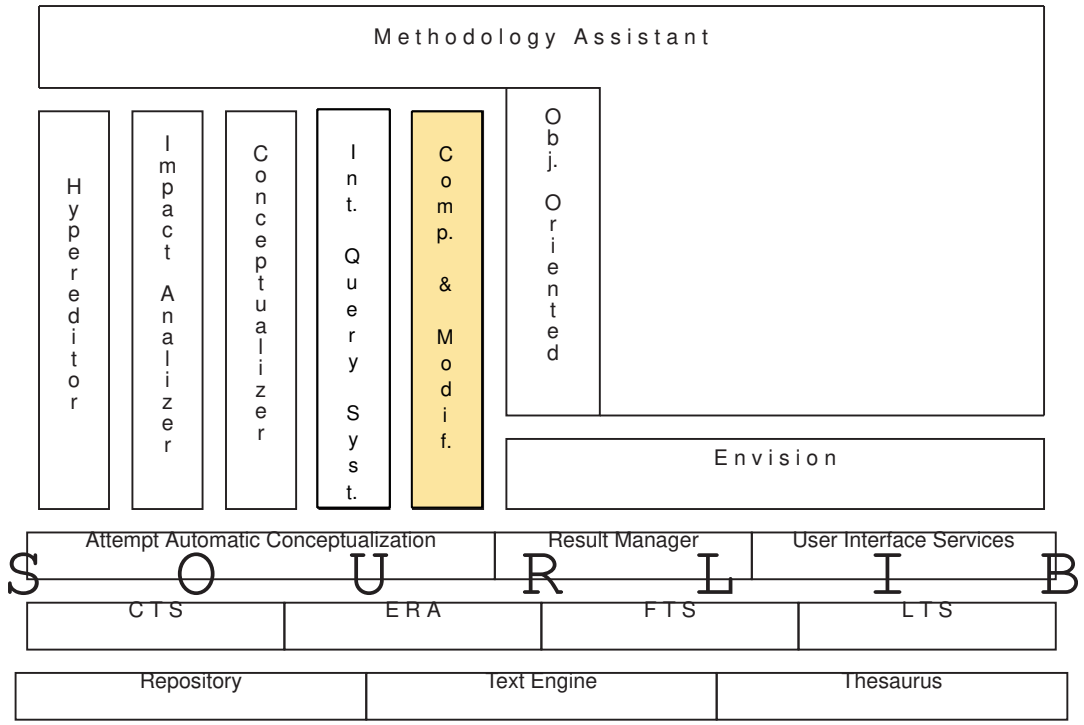


Sour'94 — Advanced Techniques for Software Reuse
15–17 December 1994
Braga

J. Nuno Oliveira
University of Minho / INESC (Group 2361)
Braga

email: jno@di.uminho.pt

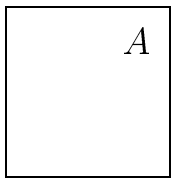
# Abstract



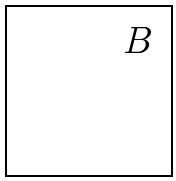

- Comparator & Modifier (CMOD) is a module central to the architecture of the SOUR software system.
- It performs crucial tasks such as comparing SOUR objects, deciding about object-equivalence, notifying the user about the impact of modifying an existing object *etc*.
- The overall design of CMOD will be presented.

# The Re-Use "Mitology"

My "new" (software) problem is...



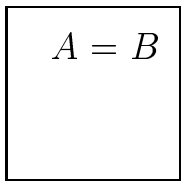

Do you know of any...




in your library which I could *reuse*?

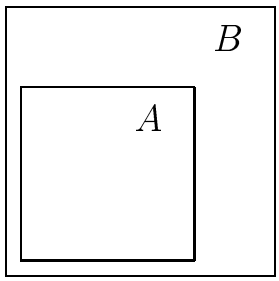
What can I expect of $B$, if a particular one is found?

- Perfect match:



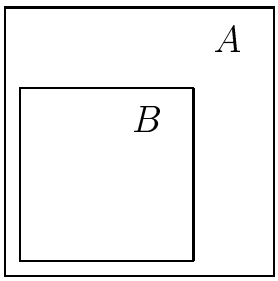

  Nothing left to do, my problem has been solved before;

- Special case:



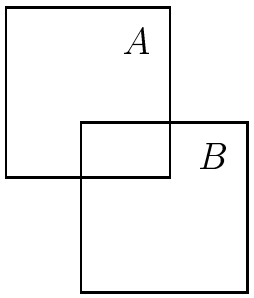

  Little to do, my problem's class has already been treated;

- General case:



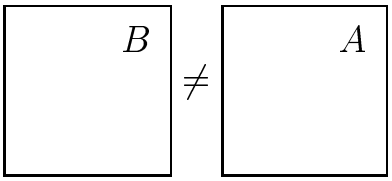

  A little effort will do: generalize somebody else's solution;

- Partial match:




  Something to do, only part of $B$ is reusable;

- Worst case:

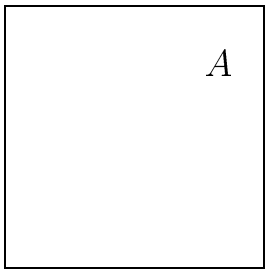
for all $B$ in library, 

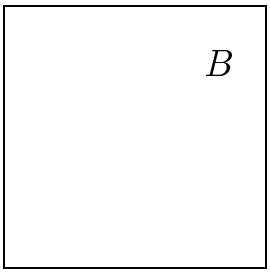

  Quite a lot to do, $A$ is really a new problem!

# What About the Past?

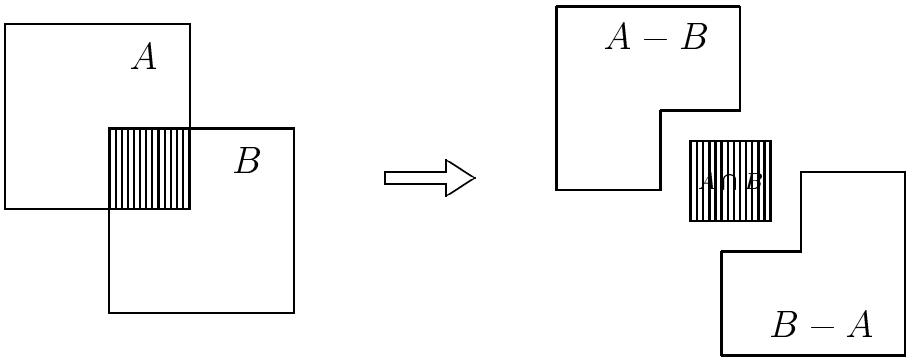
Maybe we will need a *software library "slimming diet"* algorithm:

{

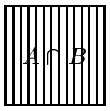
for all 

and 


in library
do




od
}

For every 

: a reusable component has been identified.

# Prieto-Diaz “Re-use Algorithm”

From references [*Prieto-Diaz & Freeman 1987*] and [*Prieto-Diaz 1991*]:

```
begin search library;
   if identical match
   then terminate
   else begin collect similar components;
            for each component in library
                compute degree of match
            end;
            rank and select best;
            modify selected component
        end
end;
```

## Questions

What does it mean...

...degree of match?

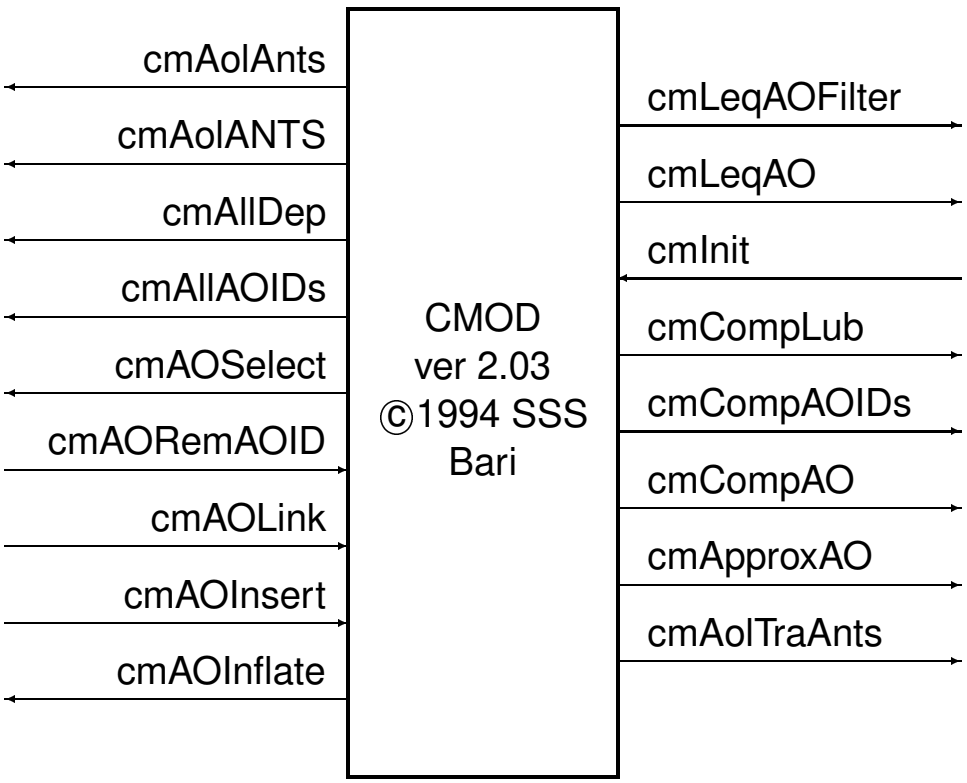
...best?

...similar?

⇒ Need for reliable software component comparison metrics!

## SOUR's Strategy in This Respect

In the SOUR project, a specific module — CMOD (Comparator & Modifier) — was designed for the purpose of managing *object comparison* and modification:



The main target of this talk is to describe the comparison metrics adopted by the `CMOD` component of SOUR.

## Questions

- What does it mean to compute $A \cap B$ in SOUR?

. . . But, first of all:

- What kind of thing "are" $A$ and $B$ in SOUR?

# About the Underlying Information Standard

- Information in the SOUR system is recorded in the form of so-called *abstract objects* (AO) which are independent of their physical support — *e.g.* textfile, POSTSCRIPT file — or location — *e.g.* pathname.
- AOs are catalogued according to an adopted *standard* of classification called *conceptualization*, factored in two layers:
  - *Coarse level* — hierarchical "is-a"-like classification, enabling conventional inherited attribute-based reasoning;
  - *Fine level* — a *multifaceted* classification schema based on fuzzy concept-network reasoning which extends conventional keyword-oriented classification.

(Note some simplifications: ignoring actions and compound objects por the moment.)

**(See picture on the next slide)**

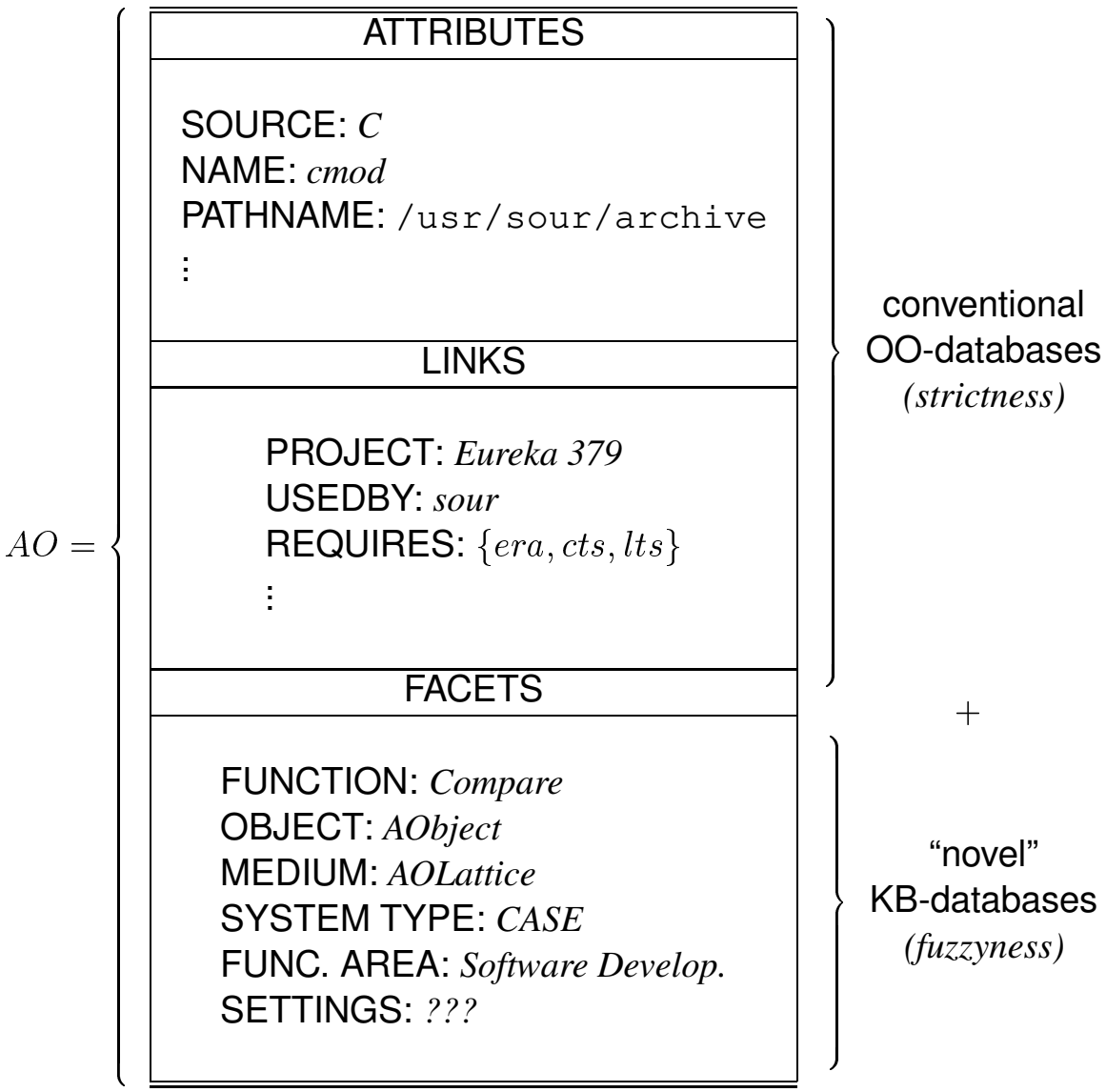
# Warning!

- AO-comparison metrics unlikely to be linear such as *e.g.*

$$1 < 2 < 3 < \ldots < i < i + 1 < \ldots < max$$

- Thus

**No best component**




# Properties We Want

(Read "$\texttt{cmCompAO}(A, B)$" as $A \cap B$ for our case, the cornerstone function of COMPARATOR's algebra):

- the comparison of an AO with itself should yield itself:

$$\texttt{cmCompAO}(x, x) = x$$

- AO-comparison should not depend on argument-order:

$$\texttt{cmCompAO}(x, y) = \texttt{cmCompAO}(y, x)$$

- AO-comparison should be compositional and associative:

$$\texttt{cmCompAO}(x, \texttt{cmCompAO}(y, z)) = \texttt{cmCompAO}(\texttt{cmCompAO}(x, y), z)$$

- a "most abstract" AO, $\top$, generalizes any other AO:

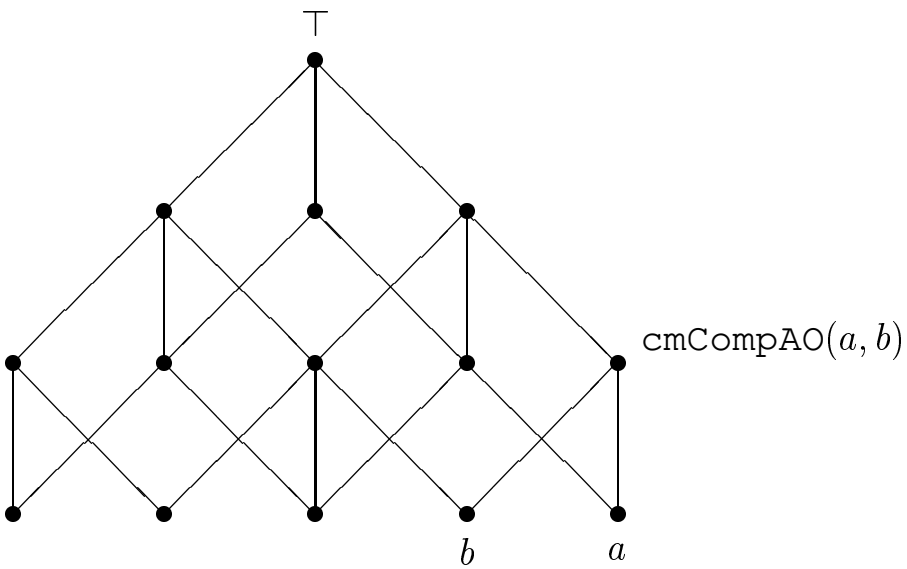
$$\texttt{cmCompAO}(\top, x) = \texttt{cmCompAO}(x, \top) = \top$$

that is,

$\top =$

| ATTRIBUTES |
|---|
| LINKS |
| FACETS |

AOs will make up a *specialize/generalize* hierarchy, with $\top$ at the top:



$a \leq b$ means $b = \texttt{cmCompAO}(a, b)$

# AO-Comparison Definition

$$\text{AO-comparison} = \begin{cases} \text{compare attributes} \rightarrow \text{strict comparison} \\ \text{compare links} \rightarrow \textit{sharp set} \text{ comparison} \\ \text{compare facets} \rightarrow \textit{fuzzy set} \text{ comparison} \end{cases}$$

## In detail,

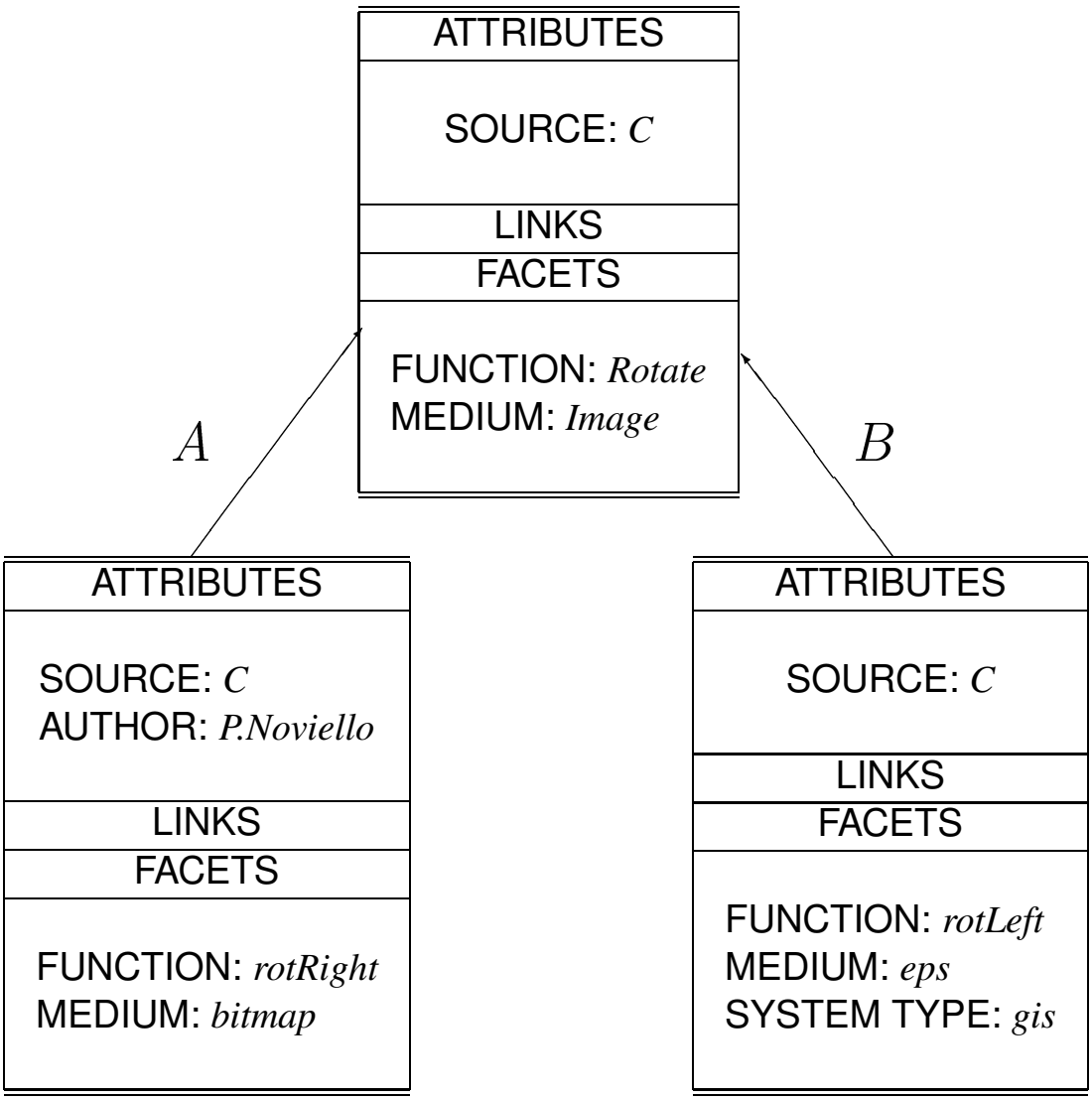
$\texttt{cmCompAO}(x, y) =$

| ATTRIBUTES |
|---|
| Every common attribute of both $x$ and $y$, bearing the same value, is selected |
| **LINKS** |
| For every common link-name of both $x$ and $y$, select only links with same target object |
| **FACETS** |
| For each common facet type consult CTS to find closest concept which unambiguously unifies the corresponding terms |

## In summary:

$\texttt{cmCompAO}(x, y) =$ generalization of both $x$ and $y$ which preserves, *as much as possible*, the common information between $x$ and $y$.

**Example:** $C = \texttt{cmCompAO}(A, B)$



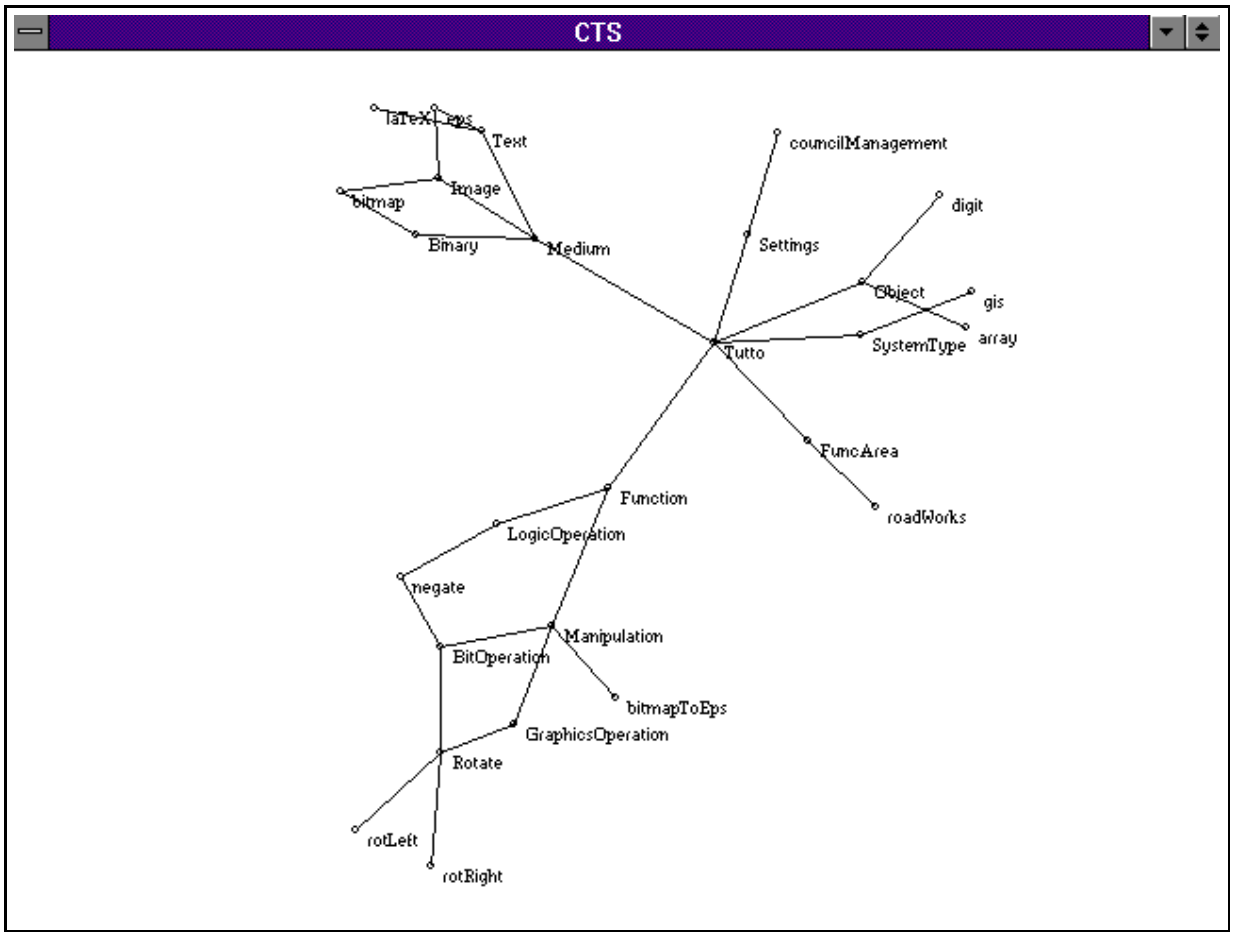

A = Piece of C code written by P. Noviello for right-rotating bitmaps.

B = Piece of C code which right-rotates encapsulated POSTSCRIPT files written for a geographical information system package.

C = piece of C code for rotating images.

# The Rôle of Fuzzyness in AO-comparison

CTS is a directed graph of terms and concepts, *e.g.*




but a special one — it is a *fuzzy graph*. What does this mean?

## We know that a graph is a set of pairs...

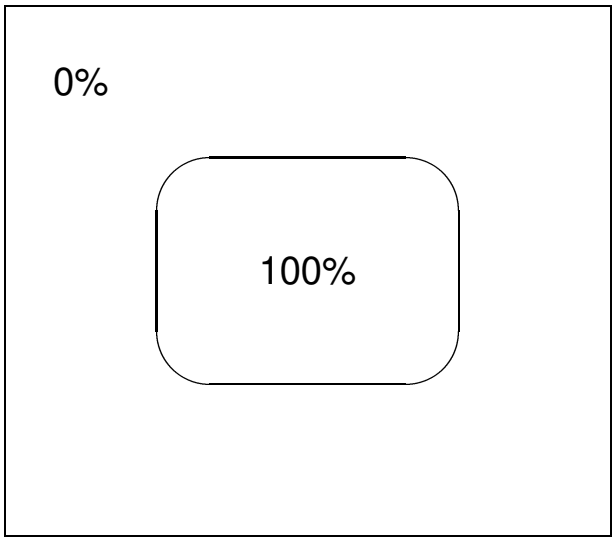
$$Graph = Term \times Term \rightarrow \{0, 1\}$$

## Generalization

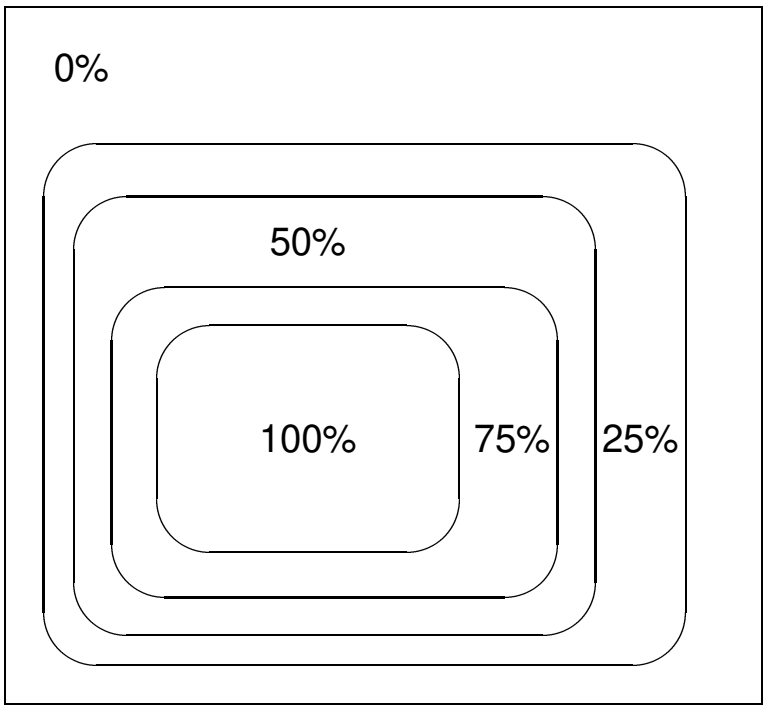
A *fuzzy graph* is a *fuzzy set* of pairs,

$$FuzzyGraph = Term \times Term \rightarrow [0, 1]$$

## Picture of a set:




## Picture of a *fuzzy set*:



In a fuzzy-graph, the “knowledge” captured by some arcs is more “certain” that others.

Example: $\boxed{add \xrightarrow{70\%} ArithOp}$ versus $\boxed{add \xrightarrow{30\%} Insertion}$.

Facet fuzzyness adds value to conventional strict comparison and prevents from too “flat” AO-hierarchies.

# About Modifier Functionality

Main purpose:

Overall task of managing the *reuse overhead* based on the AO-comparison functionality of Comparator.

In particular:

- Manage the generalize/specialize AO hierarchy in the face of AO-insertion, modification and deletion.
- Impact analysis — broadcasting of AO-modification or deletion
- Autonomization of re-usable components
- Last but not least: managing AO-equivalence.

In fact:

**AO-equivalence**

$x \leq y \wedge y \leq x$ means that two different objects $x$ and $y$ may exist in the repository which the comparison metrics cannot distinguish.

**CONCERN: Save ERA from redundant links!**

# Main API Functions

`cmAOInsert`$(a)$ — extends "normal" AO-insertion of $a$ by ERA

`cmAORemAOID`$(a)$ — extends "normal" AO-deletion of $a$ in ERA

`cmAOAllDep`$(a)$ — retrieves all objects likely to be affected by the modification of $a$

# Brief Illustration

For simplicity, assume AOs have only numerical attributes $a, b, c, \ldots$.

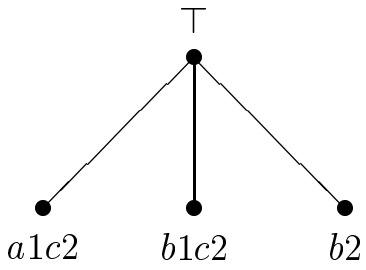
Use *e.g.* string $a1b3c7$ as a shorthand of

| ATTRIBUTES |
| --- |
| a: 1<br>b: 3<br>c: 7 |
| LINKS |
| FACETS |

Suppose starting AO-hierarchy containing $\top$ only. Execute:

$$\texttt{cmAOInsert}(a1c2);$$
$$\texttt{cmAOInsert}(b1c2);$$
$$\texttt{cmAOInsert}(b2);$$

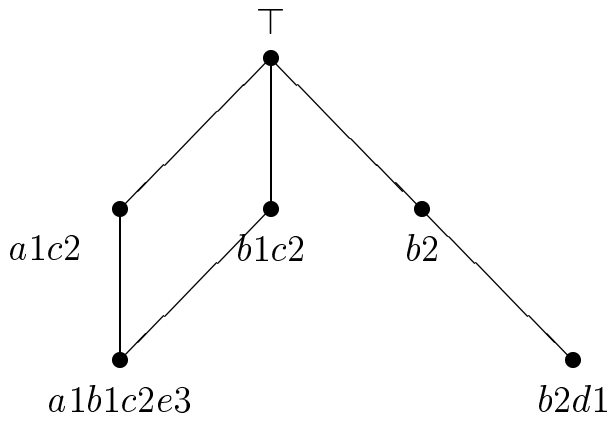
Resulting **specialize**-linked AO-hierarchy:




Now execute:

$$\texttt{cmAOInsert}(b2d1);$$
$$\texttt{cmAOInsert}(a1b1c2e3);$$

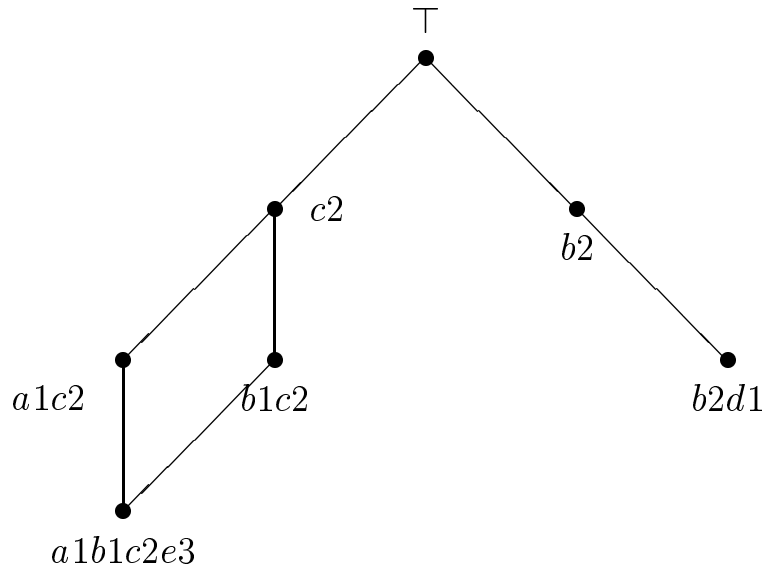
Resulting **specialize**-linked AO-hierarchy:




Now you want to autonomize the comparison between $a1c2$ and $b1c2$:

$$\texttt{cmCompAO}(a1c2, b1c2) \;=\; c2$$

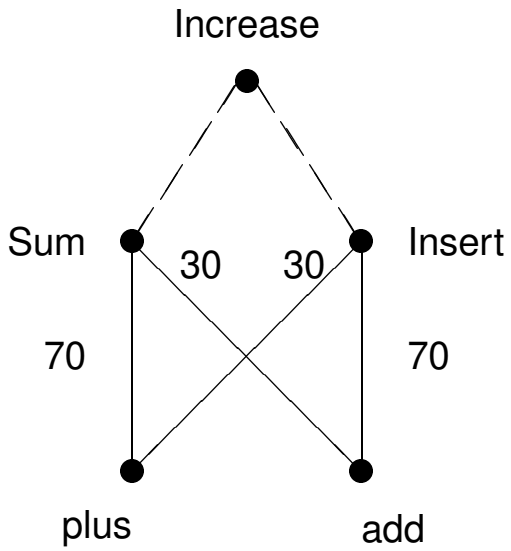
The links from $a1c2$ and $b1c2$ to $\top$ become redundant:




and so on . . . !

# Impact on Overall Design

**CTS** — CMOD imposes "mild" restrictions on facet fuzzyness:




**CON** — positive impact of CMOD while controling redundant AOs.

**RM** — AO-lattice adds to the graphical display/browsing of objects.

**IQS** — query evaluation can be assisted/optimized via CMOD.

# Final Remarks

- Performance of CMOD functionality
- AO-class-tuned dedicated comparison metrics overriding default one
- "plug-in/plug-out" CMOD

# References

**[Prieto-Diaz & Freeman 1987]** Prieto-Diaz R., Freeman P. *Classifying Software for Reusability.* IEEE, Software, 4-1, January 1987.

**[Prieto-Diaz 1991]** Prieto-Diaz R. *Implementing Faceted Classification for Software Reuse.* Comm. of the ACM, 34(5):89–97, May 1991.

**[Oliveira & Cruz 93]** Oliveira J.N., Cruz A.M. *Formal Calculi Applied to Software Component Knowledge Elicitation*. Technical Report C19-WP2d, Project *C.1.9. Sviluppo di Metodologie, Sistemi e Servizi Innovativi in Rete*, December 1993.